A Model-based Approach for Diagnosing Faults in Web Service Processes

Yuhong Yan, Philippe Dague, Yannick Pencolé, Marie-Odile Cordier

A Model-based Approach for Diagnosing Faults in Web Service Processes

Yuhong Yan[1], Philippe Dague[2], Yannick Pencolé[3], and Marie-Odile Cordier[4]
[1]National Research Council, 46 Dineen Drive, Fredericton, NB E3B 5X9, Canada
[2]LRI, Univ. Paris-Sud 11, CNRS, F-91893 Orsay, France
[3]LAAS-CNRS, F-31077 Toulouse Cedex, Fance
[4]IRISA, Campus de Beaulieu, F-35042 Rennes Cedex, France
Email: yuhong.yan@nrc.gc.ca

Abstract

Web services based on a service-oriented architecture framework provide a suitable technical foundation for business process management and integration. A business process can be composed of a set of Web services that belong to different companies and interact with each other by sending messages. Web service orchestration languages are defined by standard organizations to describe business processes composed of Web services. A business process can fail for many reasons, such as faulty Web services or mismatching messages. It is important to find out which Web services are responsible for a failed business process because we could penalize these Web services and exclude them from the business process in the future. In this paper, we propose a model-based approach to diagnose the faults in a Web service-composed business process. We convert a Web service orchestration language, more specifically BPEL4WS, into synchronized automata, so that we have a formal description of the topology and variable dependency of the business process. After an exception is thrown, the diagnoser can calculate the business process execution trajectory based on the formal model and the observed evolution of the business process. The faulty Web services are deduced from the variable dependency on the execution trajectory. We demonstrate our diagnosis technique with an example.

1 Introduction

Web services not only function as middleware for application invocation and integration, but also function as a modeling and management tool for business processes. In a Service Oriented Architecture paradigm, a business process can be composed of Web services distributed over the Internet. This kind of business processes can be flexible and optimal by using the best services from multiple companies. Various Web service process description languages are designed by standard bodies and companies. Among them, Business Process Execution Language for Web Service (BPEL4WS, denoted as

BPEL after) (Andrews, Curbera, Dholakia, Goland, & et.al., 2003) is the de facto standard used to describe an executable Web service process. In this paper, we study the behavior of a business process described in BPEL.

As any other systems, a business process can fail. For a Web service process, the symptom of a failure is that exceptions are thrown and the process halts. As the process is composed of multiple Web services, it is important to find out which Web services are responsible for the failure. If we could diagnose the faulty Web services, we could penalize these Web services and exclude them from the business process in the future. The current throw-and-catch mechanism is very preliminary for diagnosing faults. It relies on the developer associating the faults with exceptions at design time. When an exception is thrown, we say certain faults occur. But this mechanism does not guarantee the soundness and the completeness of diagnosis.

In this paper, we propose a model-based approach to diagnose faults in Web service processes. We convert the basic BPEL activities and constructs into synchronized automata whose *states* are presented by the values of the *variables*. The process changes from one state to another by executing an *action*, e.g. assigning variables, receiving or emitting messages in BPEL. The emitting messages can be a triggering *event* for another service to take an action. The diagnosing mechanism is triggered when exceptions are thrown. Using the formal model and the runtime observations from the execution of the process, we can reconstruct the unobservable trajectories of the Web service process. Then the faulty Web services are deduced based on the variable dependency on the trajectories. Studying the fault diagnosis in Web service processes serves the ultimate goal of building self-manageable and self-healing business processes.

This paper is organized as follows: section 2 analyzes the fault management tasks in Web service processes and motivates the use of Model-based Diagnosis (MBD) for Web services monitoring and diagnosis; section 3 presents the principles for MBD; section 4 formally defines the way to generate an automaton model from a BPEL description; section 5 extends the existing MBD techniques for Web service monitoring and diagnosis; section 6 is the related work, and section 7 is the conclusion.

2 Advanced Fault Management for Web Service Processes

A Web service process can run down for many reasons. For example, a composed Web service may be faulty, an incoming message mismatches the interface, or the Internet is down. The *symptom*[1] of a failed Web service process is that *exceptions* are thrown and the process is halted. The current fault handling mechanism is throw-and-catch, similar to programming languages. The *exceptions* are thrown at the places where the process cannot be executed. The *catch* clauses process the exceptions, normally to recover the failure effects by executing predefined actions.

The throw-and-catch mechanism is very preliminary for fault diagnosis. The ex-

[1] In diagnosis concept, *sympton* is an observed abnormal behavior, while *fault* is the original cause of a sympton. For example, an alarm from a smoke detector is a symptom. The two possible faults, a fire or a faulty smoke detector, are the causes of the symptom.

ception reports where it happened and returns some fault information. The exceptions can be regarded as associated with certain faults. When an exception is thrown, we deduce that its associated fault occurred. Customized exceptions are especially defined for this purpose. This kind of association relations rely on the empirical knowledge of the developer. It may not be a real cause of the exceptions. In addition, there may exist multiple causes of an exception which are unknown to the developer. Therefore, the current throw-and-catch mechanism does not provide sound and complete diagnosis. For example, when a Web service throws an exception about a value in a customer order, not only the one that throws the exception may be faulty, but the one that generates these data may also be faulty. But a Web service exception can only report the Web service where the exception happens with no way to know who generated these data. In addition, all the services that modified the data should be also suspected. Not all of this kind of reasoning is included in the current fault handling mechanism. *A systematic diagnosis mechanism which is based on the model of the Web service process and a solid theoretical foundation needs to be developed.* This is the objective of this paper.

The diagnosis task is to determine the Web services responsible for the exceptions. These Web services will be diagnosed faulty. During the execution of a BPEL process, the exceptions come from the BPEL engine or the infrastructure below, e.g. Apache Tomcat, and Internet. We classify the exceptions into ***time-out*** exceptions and ***business logic*** exceptions.

The *time-out* exceptions are due to either a disrupted network or unavailable Web services. If there is a lack of response, we cannot distinguish whether the fault is in the network or at the remote Web service, except if information is transmitted by the network fault management in the first case. Since we cannot diagnose which kind of faults prevent a Web service from responding, we can do little with *time-out* exceptions. Indeed what can be done is more statistics at the level of process classes (and not process instances) that will be used by experts to improve the QoS.

The *business logic* exceptions occur while invoking an external Web service and executing BPEL internal activities. For example, mismatching messages (including the type of parameters and the number of parameters mismatching) cause the exceptions to be thrown when the parameters are passed to the remote method. BPEL can throw exceptions indicating the input data is wrong. During execution, the remote service may stop if it cannot process the request. The most common scenarios are the invalid format of the parameters, e.g. the data is not in a valid format, and the data is out of the range. The causes of the exceptions are various and cannot be enumerated. The common thread is that a business logic exception brings back information on the variables that cause the problem. In this paper, our major effort is on diagnosing business logic-related exceptions at the process instances level.

The advanced fault management mechanism serves the ultimate goal to build self-manageable Web service processes. Fault management mechanisms can be among other self-manageable functions. Some functions related to fault management are:

- **Monitoring** the execution of Web service process, and record necessary and sufficient information for online/offline diagnosis. Insufficient information cannot produce correct diagnosis. In Web service processes, we need to keep a chronological record for some of the variables.

- **Detecting** faulty behavior. In other physical tasks, detecting needs to compare the observations with the predictions from the system description to discover the discrepancies. For Web service processes, this task is a trivial one to observe exceptions. But we can imagine to build new detectors in order to detect symptoms earlier and "closer" to the causes.
- **Diagnosing** the causes of exceptions. This is the major focus of this paper. See Section 5 for detail.
- **Recovering** from the failure effects. BPEL uses predefined compensation handlers and fault handlers to eliminate failure effects. As failure effects cannot be revealed by the empirical diagnosis mechanism in BPEL, the predefined compensation actions may not be sufficient. A more advanced recovery mechanism has to be defined, based on the model-based diagnosis developed in this paper, although it is not covered in this paper.

3 The Principle of Model-based Diagnosis for Discrete Event Systems

MBD is used to monitor and diagnose both static and dynamic systems. It is an active topic in both Artificial Intelligence (AI) and Control Theory communities. Automated diagnosis has been applied to all kinds of systems, such as communication systems, plant processes and automobiles. The early results in MBD are collected in (Hamscher, Console, & de Kleer, 1992). Let us briefly recall the terminology and notations adopted by the model-based reasoning community.

- SD: system description. In the AI-rooted diagnostic techniques, SD is symbolically modeled, e.g. in first-order logic sentences, and in DES as used in this paper.
- $COMPS$: a finite set of constants to represent the components in a system.
- *System*: a pair (SD, $COMPS$).
- D: a mode assignment to each component in the system. An assignment to a component is a unary predicate. For example, for a component $c_i \in COMPS$, $\neg ab(c_i)$ means c_i working properly, and $ab(c_i)$ means c_i is in an abnormal mode. Obviously a component has different behavior for different modes.
- *Observables*: the variables that can be observed/measured. For a physical system, the observables are the variables measured by sensors, or events reported by alarms, etc.
- OBS: a set of observations. They are the values of the *Observables*. They can be a finite set of first-order sentences, e.g. value assignments to some variables.
- *Observed system*: (SD, $COMPS$, OBS).

Diagnosis is a procedure to determine which components are correct and which components are faulty in order to be consistent with the observations and the system description. Therefore, logically, a consistency-based *diagnosis* is:

Definition 1 *D is a consistency-based **diagnosis** for the observed system $\langle SD, COMPS, OBS\rangle$, if and only if it is a mode assignment and $SD \cup D \cup OBS \nvDash \bot$.*

From Definition 1, diagnosis is a mode assignment D that makes the union of SD, D and OBS logically consistent. D can be partitioned into two parts:

- D_{ok} which is the set of components which are assigned to the $\neg ab$ mode;
- D_f which is the set of components which are assigned to the ab mode.

Usually we are interested in those diagnoses which involve a minimal set of faults, i.e., the diagnoses for which D_f is minimal for set inclusion.

Definition 2 *A diagnosis D is **minimal** if and only if there is no other diagnosis D' for $\langle SD, COMPS, OBS\rangle$ such that $D'_f \subset D_f$.*

The dual concept of a diagnosis is a conflict.

Definition 3 *A set $CO \subseteq COMPS$ is a conflict for $\langle SD, COMPS, OBS\rangle$, if and only if $SD \cup OBS \cup \{\neg ab(C) | C \in CO\} \models \bot$.*

Similarly a minimal conflict is a conflict that is minimal for set inclusion. In (Reiter, 1987), Reiter introduces the hitting set algorithm for computing minimal diagnoses using the set of conflicts.

Definition 4 *((Reiter, 1987)) Let C be a collection of sets. A hitting set for C is a set $H \subseteq \bigcup_{S \in C} S$ such that $H \cap S \neq \emptyset$ for each $S \in C$. A hitting set is minimal if no proper subset of it is a hitting set.*

Theorem 1 *((Reiter, 1987)) A set $D \subseteq COMPS$ is a minimal diagnosis for $\langle SD, COMP, OBS\rangle$ if and only if D is a minimal hitting set for the collection of conflicts (or equivalently for the collection of minimal conflicts).*

When the system description is in first order logic, the computation of all diagnoses is more generally rooted in automated reasoning, relying on prime implicates of $SD \cup OBS$ in the form of disjuncts of ab-literals, and on their prime implicants in the form of conjuncts of ab-literals (Hamscher et al., 1992).

When applying MBD, a formal system description is needed. Therefore, we need to study the proper formal model for Web service processes. As the interactions between Web services are driven by message passing, and message passing can be seen as discrete events, we consider the Discrete Event Systems (DES) suitable to model Web service processes. Many discrete event models, such as Petri nets, process algebras and automata, can be used for Web service process modeling. These models were invented for different purposes, but now they share many common techniques, such as symbolic

representation (in addition to graph representation in some models) and similar symbolic operations. In this paper, we present a method to represent Web service processes described in BPEL as automata in Section 4. Here we introduce MBD techniques for automata. A classic definition of deterministic automaton is as below:

Definition 5 *An **automaton** Γ is a tuple $\Gamma = \langle X, \Sigma, T, I, F \rangle$ where:*

- *X is a finite set of states;*
- *Σ is a finite set of events;*
- *$T \subseteq X \times \Sigma \rightarrow X$ is a finite set of transitions;*
- *$I \subseteq X$ is a finite set of initial states;*
- *$F \subseteq X$ is a finite set of final states.*

Definition 6, 7 and 8 are some basic concepts and operations about automata.

Definition 6 ***Synchronization*** *between two automata $\Gamma_1 = \langle X_1, \Sigma_1, T_1, I_1, F_1 \rangle$ and $\Gamma_2 = \langle X_2, \Sigma_2, T_2, I_2, F_2 \rangle$, with $\Sigma_1 \cap \Sigma_2 \neq \emptyset$, produces an automaton $\Gamma = \Gamma_1 \| \Gamma_2$, where $\Gamma = \langle X_1 \times X_2, \Sigma_1 \cup \Sigma_2, T, I_1 \times I_2, F_1 \times F_2 \rangle$, with:*
$T((x_1, x_2), e) = (T_1(x_1, e), T_2(x_2, e))$, *if* $e \in \Sigma_1 \cap \Sigma_2$
$T((x_1, x_2), e) = (T_1(x_1, e), x_2)$, *if* $e \in \Sigma_1 \backslash \Sigma_2$
$T((x_1, x_2), e) = (x_1, T_2(x_2, e))$, *if* $e \in \Sigma_2 \backslash \Sigma_1$

Assume $s = \Sigma_1 \cap \Sigma_2$ is the joint event set of Γ_1 and Γ_2, Γ can also be written as $\Gamma = \Gamma_1 \|_s \Gamma_2$.

Example 1 *In Figure 1, Γ_1 and Γ_2 are two automata. The third one Γ_3 is produced by synchronizing Γ_1 and Γ_2.*

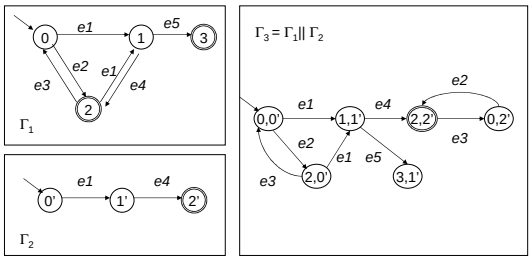


Figure 1: An example of synchronization

Definition 7 *A **trajectory** of an automaton is a path of contiguous states and transitions in the automaton that begins at an initial state and ends at a final state of the automaton.*

Example 2 *The trajectories in the automaton Γ_3 in Figure 1 can be represented as the two formulas below, in which $[\ ]^*$ means the content in $[\ ]$ repeated 0 or more times:*

$[(0,0')\overset{e2}{\rightarrow}(2,0')\overset{e3}{\rightarrow}]^*[(0,0')\overset{e1}{\rightarrow}(1,1')\overset{e4}{\rightarrow}(2,2')][\overset{e3}{\rightarrow}(0,2')\overset{e2}{\rightarrow}(2,2')]^*$,
$[(0,0')\overset{e2}{\rightarrow}(2,0')\overset{e3}{\rightarrow}]^*[(0,0')\overset{e2}{\rightarrow}(2,0')\overset{e1}{\rightarrow}(1,1')\overset{e4}{\rightarrow}(2,2')][\overset{e3}{\rightarrow}(0,2')\overset{e2}{\rightarrow}(2,2')]^*$.

Definition 8 ***Concatenation*** *between two automata $\Gamma_1 = \langle X_1, \Sigma_1, T_1, I_1, F_1\rangle$ and $\Gamma_2 = \langle X_2, \Sigma_2, T_2, I_2, F_2\rangle$, with $\Sigma_1 \cap \Sigma_2 = \emptyset$ and $F_1 \cap I_2 \neq \emptyset$, produces an automaton $\Gamma = \Gamma_1 \circ \Gamma_2$, where $\Gamma = \langle X_1 \cup X_2, \Sigma_1 \cup \Sigma_2, T_1 \cup T_2, I_1, F_2 \cup (F_1 \backslash I_2)\rangle$.*

The principle of diagnosis using DES models was founded by (Sampath, Sengupta, Lafortune, Sinnamohideen, & Teneketzis, 1995) and (Cordier & Thiébaux, 1994). System description SD models both correct and faulty behavior of a system. Assume system description SD is an automaton Γ, and observed events in chronological order are represented as another automaton OBS. Assume the joint event set of Γ and OBS is s. In this context, we call Diagnosis the automaton produced by synchronizing Γ and OBS:

$$\mathsf{Diagnosis} = \Gamma||_s OBS \tag{1}$$

From the definition of synchronization, it is easy to prove that each trajectory in Diagnosis explains the sequence of observations in the sense that observable events in the trajectory occur in the identical chronological order as in OBS, i.e.:

$$\mathsf{Diagnosis} \models OBS \tag{2}$$

Therefore, Diagnosis for DES is what is called an *abductive diagnosis* in MBD theory.

Example 3 *In Figure 1, Γ_1 is a system description in which e_2 and e_3 represent occurrences of faults which are not observable directly (otherwise, the diagnosis would be trivial). Γ_2 is an observation in which two events e_1 and e_4 are observed sequentially. The* Diagnosis *is Γ_3.*

It is not so easy to compute the trajectories of Diagnosis because there are several possibilities for trajectory expansion that can arise from partial observations. We need to get all the possible trajectories. For trajectory expansion, people basically use search algorithms. Other algorithms, rooted from search algorithms, can also be used. For example, planning tools and model checking tools are used for trajectory expansion. Of course, these tools have to be modified in order to get complete trajectories.

Diagnostic process is almost achieved when Diagnosis is obtained, because Diagnosis explains the observations based on SD (as an automaton Γ). If we want to obtain diagnoses $\{\mathrm{D}\}$ as mode assignments as in the consistency-based framework, we need a mapping function f : Diagnosis $\mapsto \{D\}$. Each trajectory t in Diagnosis is mapped into a D, i.e. $t \mapsto D$. As domain knowledge, a faulty event e_f is known to be associated with a fault mode $F(c_i)$ of some component c_i, i.e. $e_f \leftrightarrow F(c_i)$[2]. If

[2] $F(c_i)$ is a specific fault mode. When we do not know a specific fault mode, we use $ab(c_i)$ to represent c_i is faulty.

e_f is included in a trajectory t, we deduce that the correspondent fault $F(c_i)$ occurs. Formally,

Proposition 1 *Assume t is a trajectory in* Diagnosis*, then* $t \mapsto D$ *where mode assignment D is defined by* $D_f = \{c_j | e_f \leftrightarrow F(c_j) \text{ and } e_f \in t\}$ *(and thus* $D_{ok} = \{c_j | c_j \in COMPS \backslash D_f\}$*.*

As each fault event maps to a fault, practically we need only to know the set of faulty events in a trajectory:

$$t \mapsto \{e_f | e_f \in t\} \tag{3}$$

From (3), if we know $\{e_f\}$, we can easily get D_f and thus D. In the following, we use $\{e_f\}$ to represent a D_f. As there are often multiple trajectories $\{t^i\}$ in Diagnosis, the diagnoses $\{D^i\}$ are also multiple:

Proposition 2 *Assume* $\{t^i\}$ *is the set of all trajectories in* Diagnosis*, then* $\{t^i\} \mapsto \{D^i\}$*, where* $D^i_f = \{c_j | e^i_f \leftrightarrow F(c_j)$ *and* $e^i_f \in t^i\} \subseteq D^i$*.*

In general, we are interested only in minimal diagnoses, i.e. in Proposition 2 we keep only those D^i_f which are minimal.

Example 4 *From* Diagnosis Γ_3 *in Figure 1, we get 2 kinds of possible sequences of faulty events:*

$$\{[e2, e3]^*, [e3, e2]^*\}, \{[e2, e3]^*, e2, [e3, e2]^*\}.$$

From the above sequences, we can get three diagnoses:

$$\{\}, \{e2\}, \{e2, e3\}.$$

The minimal diagnosis is $\{\}$*, which means no fault.*

In Example 4, different trajectories give us different diagnoses. It can be no faults, or $e2$ (mapped to its fault), or both $e2$ and $e3$. They are all sound. Adding more observables is a way to clarify the ambiguity. To determine the observables for diagnosing a certain fault is the problem of diagnosability which is not covered in this paper. Below is another example without ambiguity:

Example 5 *In Figure 2,* Γ_1 *is* SD *and* Γ_2 *is* OBS*.* Γ_3 *is* Diagnosis*. Since* e_3 *is within the only trajectory, we can deduce that a fault represented by* e_3 *occurred.*

We need to point out that the existing diagnosis methods for physical systems modeled as DES are not in general suitable for Web service processes. First, we cannot enumerate faults in Web service environments because we do not know how a Web service can be faulty if it belongs to another company. Second, it is relatively easy to keep a record for how the software is executed by recording any selected variables. In contrast, it is more difficult to insert a sensor in a physical system. Therefore it is very difficult to reconstruct the trajectories for a physical system, but it is not a key issue for

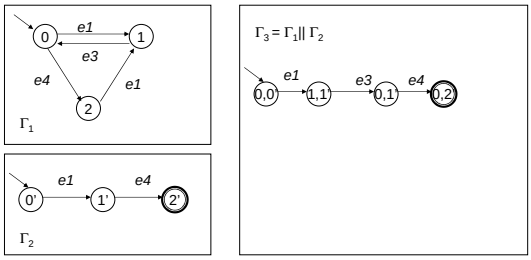


Figure 2: An example of Diagnosis

diagnosing a Web service process. We will discuss the diagnosis of Web services in Section 5.

Several advances have recently been made: the decentralized diagnoser approach (Pencolé & Cordier, 2005) (a diagnosis system based on several interacting DESs); the incremental diagnosis approach (Grastien, Cordier, & Largouët, 2005) (a monitoring system that online updates diagnosis over time given new observations); active system approaches (Baroni, Lamperti, Pogliano, & Zanella, 1999) (approaches that deal with hierarchical and asynchronized DESs); and diagnosis on reconfigurable systems (Grastien, Cordier, & Largouët, 2004). The existing techniques, such as the diagnoser approach (Pencolé, Cordier, & Rozé, 2002) or the silent closure (Baroni et al., 1999), reconstruct the unobservable behavior of the system that are required to compute diagnoses.

4 Modeling Web Service Processes with Discrete-Event Systems

4.1 Description of the Web Service Processes

BPEL is an XML-based orchestration language developed by IBM and recognized by OASIS (Andrews et al., 2003). BPEL is a so-called executable language because it defines the internal behavior of a Web service process, as compared to choreography languages that define only the interactions among the Web services and are not executable.

BPEL defines fifteen activity types. Some of them are *basic activities*; the others are *structured activities*. Among the basic activities, the most important are the following:

1. the ⟨receive⟩ activity is for accepting the triggering message from another Web service;
2. the ⟨reply⟩ activity is for returning the response to its requestor;
3. the ⟨invoke⟩ activity is for invoking another Web service.

The structured activities define the execution orders of the activities inside their scopes. For example:

- Ordinary sequential control between activities is provided by ⟨sequence⟩.
- Concurrency and synchronization between activities is provided by ⟨flow⟩.
- Loop is provided by ⟨while⟩.
- Nondeterministic choice based on external events is provided by ⟨pick⟩ and ⟨switch⟩.

Execution orders are also modified by defining the synchronization links between two activities (cf. Section 4.3.3). Normally, BPEL has one entry point to start the process and one point to exit, though multiple entry points are allowed. The variables in BPEL are actually the Simple Object Access Protocol (SOAP) messages defined in Web Service Description Language (WSDL). Therefore, the variables in BPEL are objects that have several attributes (called "parts" in WSDL).

4.2 An Example: the Loan Approval Process

Example 6 *The loan approval process is an example described in the BPEL Specification 1.1 (Andrews et al., 2003). It is diagrammed in Figure 3.*

This process contains five activities (big shaded blocks). An activity involves a set of input and output variables (dotted box besides each activity). All the variables are of composite type. The edges show the execution order of the activities. When two edges are issued from the same activity, only one edge that satisfies a triggering condition (shown on the edge) will be activated. In this example, the process is triggered when a ⟨receive⟩ activity named $receive1$ receives a message of a predefined type. First, $receive1$ initializes a variable $request$. Then, $receive1$ dispatches the request to two ⟨invoke⟩ activities, $invokeAssessor$ and $invokeApprover$, depending on the amount of the loan. In the case where the amount is large ($request.amount >= 1000$), $invokeApprover$ is called for a decision. In the case where the amount is small ($request.amount < 1000$), $invokeAssessor$ is called for risk assessment. If $invokeAssessor$ returns with an assessment that the risk level is low ($risk.level = low$), a reply is prepared by an ⟨assign⟩ activity and later sent out by a ⟨reply⟩ activity. If the risk level is not low, $invokeApprover$ is invoked for a final decision. The result from $invokeApprover$ is sent to the client by the ⟨reply⟩ activity.

4.3 Modeling Web Services Process with Discrete-Event Systems

A Web service process defined in BPEL is a composition of activities. We are going to model a BPEL activity as an automaton. A BPEL code has a finite set of variables and a BPEL state is associated with an assignment of these variables. A BPEL activity is triggered when its initial state satisfies a finite set of triggering conditions which is a certain assignment of variables. After an activity is executed, the values of the state

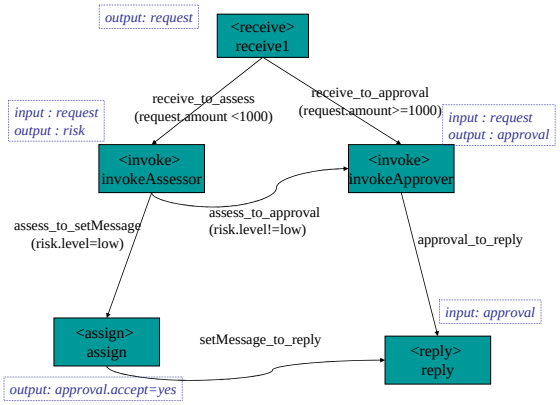


Figure 3: A loan approval process. Activities are represented in shaded boxes. The $inVar$ and $outVar$ are respectively the input and output variables of an activity.

variables are changed. We need to extend the classic automaton definition to include the operations on state variables.

Assume a BPEL process has a finite set of variables $V = \{v_1, \ldots, v_n\}$, and the domain $\mathcal{D} = \{D_1, \ldots, D_n\}$ for V is real values $\Re$ or arbitrary strings. $\mathcal{C} = \{c_1, \ldots, c_m\}$ is a finite set of constraints. A constraint c_j of some arity k is defined as a subset of the cartesian product over variables $\{v_{ji}, \ldots, v_{jk}\} \subseteq V$, i.e. $c_j \subseteq D_{j1} \times \cdots \times D_{jk}$, or a first order formula over $\{v_{ji}, \ldots, v_{jk}\}$. A constraint restricts the possible values of the k variables.

A BPEL state s is defined as an assignment of variables. A BPEL transition t is an operation on the state s_i, i.e., $(s_j, post(V_2)) = t(s_i, e, pre(V_1))$, where $V_1 \subseteq V$, $V_2 \subseteq V$, $pre(V_1) \subseteq \mathcal{C}$ is a set of preconditions that s_i has to satisfy and $post(V_2) \subseteq \mathcal{C}$ is a set of post-conditions that the successor state s_j will satisfy. In another word, the transition t is triggered only when the starting state satisfies the preconditions, and the operation of this transition results in a state that satisfies the post-conditions. If a state s satisfies a constraint c, we annotate as $c \wedge s$. Then, the semantics of transition t is also represented as:

$$t : (s_i \wedge pre(V_1)) \xrightarrow{e} (s_j \wedge post(V_2)).$$

Definition 9 *A **BPEL activity** is an automaton $\langle X, \Sigma, T, I, F, \mathcal{C} \rangle$, where $\mathcal{C}$ is a constraint set that defines states X and $T : X \times \Sigma \times 2^{\mathcal{C}} \rightarrow X \times 2^{\mathcal{C}}$.*

4.3.1 Modeling Basic Activities

In the following, we enumerate the model for each basic activity.

Activity ⟨receive⟩: $\langle \{s_o, s_f\}, \{received\}, \{t\}, \{s_o\}, \{s_f\}, \mathcal{C} \rangle$ with
$t : (s_o \wedge SoapMsg.type = MsgType) \xrightarrow{received} (s_f \wedge RecMsg = SoapMsg)$, where

$MsgType$ is a predefined message type. If the incoming message $SoapMsg$ has the predefined type, $RecMsg$ is initialized as $SoapMsg$.

Activity $\langle$reply$\rangle$: $\langle\{s_o, s_f\}, \{replied\}, \{t\}, \{s_o\}, \{s_f\}, \mathcal{C}\rangle$ with
$t : (s_o \wedge exists(RepMsg)) \xrightarrow{replied} (SoapMsg = RepMsg \wedge s_f)$, where
$exists(RepMsg)$ is the predicate checking that the replay message $RepMsg$ is initialized. $SoapMsg$ is the message on the wire.

Activity $\langle$invoke$\rangle$
Synchronous invocation: $\langle\{s_o, wait, s_f\}, \{invoked, received\}, \{t_1, t_2\}, \{s_o\}, \{s_f\}, \mathcal{C}\rangle$ with
$t_1 : (s_o \wedge exists(InVar)) \xrightarrow{invoked} (wait)$, and
$t_2 : (wait \wedge SoapMsg.type = MsgType) \xrightarrow{received} (s_f \wedge exists(OutVar))$ where $InVar$ and $OutVar$ are the input and output variables.
Asynchronous invocation: $\langle\{s_o, s_f\}, \{invoked\}, \{t\}, \{s_o\}, \{s_f\}, \mathcal{C}\rangle$ with
$t : (s_o \wedge exists(InVar)) \xrightarrow{invoked} (s_f)$, asynchronous invocation does not wait for a return message.

Activity $\langle$assign$\rangle$: $\langle\{s_o, s_f\}, \{assigned\}, \{t\}, \{s_o\}, \{s_f\}, \mathcal{C}\rangle$ with
$t : (s_o \wedge exists(InVar)) \xrightarrow{assigned} (s_f \wedge OutVar = InVar)$

Activity $\langle$throw$\rangle$: $\langle\{s_o, s_f\}, \{thrown\}, \{t\}, \{s_o\}, \{s_f\}, \mathcal{C}\rangle$ with
$t : (s_o \wedge Fault.mode = \mathit{Off}) \xrightarrow{thrown} (s_f \wedge Fault.mode = On)$

Activity $\langle$wait$\rangle$: $\langle\{s_o, wait, s_f\}, \{waiting, waited\}, \{t_1, t_2\}, \{s_o\}, \{s_f\}, \mathcal{C}\rangle$ with
$t_1 : (s_o \wedge Wait\_mode = \mathit{Off}) \xrightarrow{waiting} (wait \wedge Wait\_mode = On)$
$t_2 : (wait \wedge Wait\_mode = On) \xrightarrow{waited} (s_f \wedge Wait\_mode = \mathit{Off})$
This model is not temporal. We do not consider time, so the notion of delay is not considered in this activity.

Activity $\langle$empty$\rangle$: $\langle\{s_o, s_f\}, \{empty\}, \{t\}, \{s_o\}, \{s_f\}, \mathcal{C}\rangle$
$t : (s_o) \xrightarrow{empty} (s_f)$

4.3.2 Modeling Structured Activities

Structured activities nest other activities. We can model the structured activities as automata. Note that any automaton modeling a basic activity or a structured activity has only one initial state and one final state. In the following are the automata for the structured activities.

Sequence
A $\langle$sequence$\rangle$ can nest n activities $\langle \mathsf{A_i} \rangle$ in its scope. These activities are executed in sequential order. Assume $\langle \mathsf{A_i} \rangle : \langle S_{A_i}, \Sigma_{A_i}, T_{A_i}, \{s_{A_{io}}\}, \{s_{A_{if}}\}, \mathcal{C}_{A_i}\rangle, i \in \{1, \ldots, n\}$.

Activity $\langle$sequence$\rangle$: $\langle\{s_o, s_f\} \cup \bigcup S_{A_i}, \{end\} \cup \bigcup \{callA_i\} \cup \bigcup \Sigma_{A_i}, \{t_i\} \cup \bigcup T_{A_i}, \{s_o\}, \{s_f\}, \bigcup \mathcal{C}_{A_i}\rangle$ with

$t_0 : (s_o) \xrightarrow{callA_1} (s_{A_{1o}})$
$t_i : (s_{A_{if}}) \xrightarrow{callA_{i+1}} (s_{A_{i+1o}})$
$t_n : (s_{A_{nf}}) \xrightarrow{end} (s_f)$

If assume $s_o = s_{A_{1o}}$, $s_f = s_{A_{nf}}$, and $s_{A_{if}} = s_{A_{i+1o}}$, for $i = [1, \ldots, n-1]$, a short representation of ⟨sequence⟩ is the concatenation of the nested activities $A_1 \circ A_2 \cdots \circ A_n$.

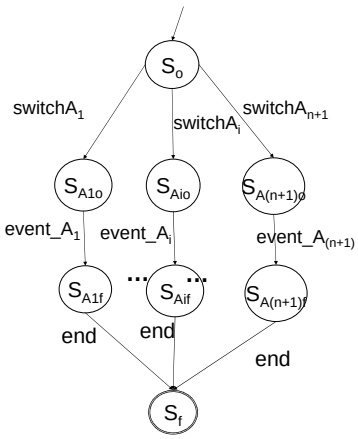
Switch

Assume a ⟨switch⟩ has n ⟨case⟩ branches and one ⟨otherwise⟩ branch (see Figure 4(a)). Assume $\langle \mathsf{A_i} \rangle : \langle S_{A_i}, \Sigma_{A_i}, T_{A_i}, \{s_{A_{io}}\}, \{s_{A_{if}}\}, \mathcal{C}_{A_i} \rangle$, $i \in \{1, \ldots, n+1\}$.

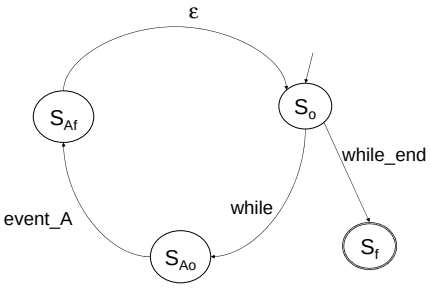
Activity ⟨switch⟩: $\langle \{s_o, s_f\} \cup \bigcup S_{A_i}, \{end\} \cup \bigcup \{switchA_i\} \cup \bigcup \Sigma_{A_i}, \bigcup \{t_{io}\} \cup \bigcup \{t_{if}\} \cup \bigcup T_{A_i}, \{s_o\}, \{s_f\}, \bigcup \mathcal{C}_{A_i} \cup \bigcup pre(V_i) \rangle$.

Assume $V_1, \ldots, V_n$ are variable sets on n ⟨case⟩ branches, $pre(V_1), \ldots, pre(V_n)$ are the constraints defined by the attributes `condition` in ⟨case⟩. The transitions are defined as below:

$t_{io} : (s_o \wedge \neg pre(V_1) \wedge \cdots \wedge pre(V_i) \cdots \wedge \neg pre(V_n)) \xrightarrow{switchA_i} (s_{A_{io}}), \forall i \in \{1, \ldots n\}$
$t_{(n+1)o}(s_o \wedge \neg pre(V_1) \wedge \cdots \wedge \neg pre(V_i) \cdots \wedge \neg pre(V_n)) \xrightarrow{swicthA_{n+1}} (s_{A_{(n+1)o}})$
$t_{if} : (s_{A_{if}}) \xrightarrow{end} (s_f), \forall i \in \{1, \ldots n+1\}$



(a)The automaton for ⟨switch⟩.



(b)The automaton for ⟨while⟩.

Figure 4: The automata for ⟨switch⟩ and ⟨while⟩

While Assume ⟨while⟩ nests an activity ⟨A⟩: $\langle S_A, \Sigma_A, T_A, \{s_{A_o}\}, \{s_{A_f}\}, \mathcal{C} \rangle$. (see Figure 4(b)).

Activity ⟨while⟩: $\langle \{s_o, s_f\} \cup S_A, \{while, while\_end\} \cup \Sigma_A, \{t_o, t_f, t\} \cup T_A, \{s_o\}, \{s_f\}, \mathcal{C} \cup pre(W) \rangle$.

Assume W is a variable set, and $pre(W)$ is the constraint defined by the attribute `condition` in ⟨while⟩.

$t_o : (s_o \wedge pre(W)) \xrightarrow{while} (s_{A_o})$

$t_f : (s_o \wedge \neg pre(W)) \xrightarrow{while\_end} (s_f)$

$t : (s_{A_f}) \xrightarrow{\epsilon} (s_o)$

Flow

A ⟨flow⟩ can nest n activities ⟨A$_i$⟩ in its scope. These activities are executed concurrently. Assume $\langle A_i \rangle : \langle S_{A_i}, \Sigma_{A_i}, T_{A_i}, \{s_{A_{io}}\}, \{s_{A_{if}}\}, \mathcal{C}_{A_i} \rangle, i \in \{1, \ldots, n\}$.

Activity ⟨flow⟩: $\langle \{s_o, s_f\} \cup \bigcup S_{A_i}, \{start, end\} \cup \bigcup \Sigma_{A_i}, \bigcup \{t_{io}, t_{if}\} \cup \bigcup T_{A_i}, \{s_o\}, \{s_f\}, \bigcup \mathcal{C}_{A_i} \rangle$ with

$t_{io} : (s_o) \xrightarrow{start} (s_{A_{io}})$

$t_{if} : (s_{A_{if}}) \xrightarrow{end} (s_f)$

Notice that the semantic of automata cannot model concurrency. We actually model the n-paralleled branches into n automata and define synchronization events to build their connections. The principle is illustrated in Figure 5. At the left, each branch is modeled as an individual automaton. The entry state s_o and the end state s_f are duplicated in each branch. Events $start$ and end are the synchronization events. At the right is the automaton resulted by synchronization. More complicated case in joining the paralleled branches is discussed in subsection 4.3.3. The key point in reasoning about decentralized automata is to postpone the synchronization until a synthesis result is needed, in order to avoid the state explosion problem (Pencolé et al., 2002)(Pencolé & Cordier, 2005). In Web service diagnosis, it is the situation (cf. subsection 5.1).

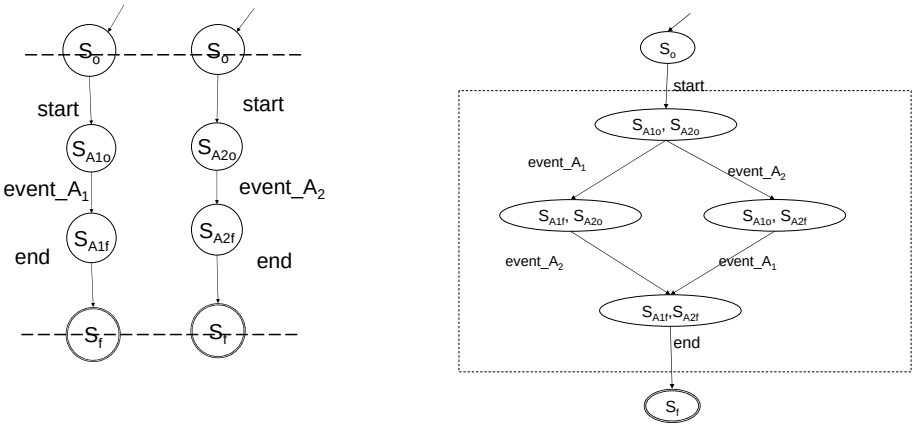
Pick

Assume a ⟨pick⟩ has n ⟨onMessage⟩ and one ⟨onAlarm⟩ branches. ⟨onMessage⟩ branches are triggered by predefined events. Assume activities $\{A_1, \ldots, A_n\}$ are corresponding to the n branches respectively. ⟨onAlarm⟩ branch is triggered by a time-out event produced by a timer. Assume activity A_{n+1} is corresponding to ⟨onAlarm⟩ branch. Exactly one branch will be selected based on the occurrence of the event associated with before any others. Assume $\langle A_i \rangle : \langle S_{A_i}, \Sigma_{A_i}, T_{A_i}, \{s_{A_{io}}\}, \{s_{A_{if}}\}, \mathcal{C}_{A_i} \rangle$, $i \in \{1, \ldots, n+1\}$.

Activity ⟨pick⟩: $\langle \{s_o, s_f\} \cup \bigcup S_{A_i}, \bigcup \{start_{A_i}\} \cup \{end\} \cup \bigcup \Sigma_{A_i}, \bigcup \{t_{io}, t_{if}\} \cup \bigcup T_{A_i}, \{s_o\}, \{s_f\}, \bigcup \mathcal{C}_{A_i} \cup \bigcup exists(event_{A_i}) \rangle$ with

$t_{io} : (s_o \wedge exists(event_{A_i})) \xrightarrow{start_{A_i}} (s_{A_{io}})$

$t_{if} : (s_{A_{if}}) \xrightarrow{end} (s_f)$



(a) (b)

Figure 5: Build concurrency as synchronized DES pieces. (a) Concurrency branches for DES pieces; (b) The joint DES model.

4.3.3 Synchronization Links of Activities

Each BPEL activity can optionally nest the standard elements ⟨source⟩ and ⟨target⟩. The XML grammar is defined as:

$< source\ linkName = \text{“}ncname\text{”}\ transitionCondition = \text{“}bool - expr\text{”}?/ >$
$< target\ linkName = \text{“}ncname\text{”}/ >$

A pair of ⟨source⟩ and ⟨target⟩ defines a link which connects two activities. The target activity must wait until the source activity finishes. Therefore, links define the sequential orders of activities. When one ⟨flow⟩ contains two parallel activities which are connected by a link, the two activities become sequentially ordered. An activity may have multiple ⟨source⟩ or ⟨target⟩ elements. Links can express richer logics, but they make the processes more difficult to analyse.

⟨source⟩ can be modeled similarly like an ⟨activity⟩, with "*transitionCondition*" as the triggering condition.

Activity ⟨source⟩: $\langle\{s_o, s_f\}, \{\epsilon\}, \{t\}, \{s_o\}, \{s_f\}, transitionCondition\rangle$ with
$t : (s_o \wedge transitionCondition) \xrightarrow{\epsilon} (s_f)$,

When an activity is the ⟨target⟩ of multiple links, a join condition is used to specify how these links can join. The join condition is defined within the activity. BPEL specification defines standard attributes for this activity:

$< activityName = \text{“}ncname\text{”}, joinCondition = \text{“}bool - expr\text{”},$
$suppressJoinFailure = \text{“}yes|no\text{”}/ >$

where $joinCondition$ is the logical OR of the liveness status of all links that are targeted at this activity. If the condition is not satisfied, the activity is bypassed, and a fault is thrown if $suppressJoinFailure$ is no.

In this case, the synchronization event end as in Figure 5(a) is removed. If the ending state of ⟨flow⟩ is the starting state s'_o of the next activity, the precondition of s'_o is the $joinCondition$. For example, either of the endings of the two branches can trigger the next activity can be represented as: $s'_o \wedge (exists(s_{A_{1f}}) \vee exists(s_{A_{2f}}))$.

4.4 Modeling the Loan Approval Process

In this section, we present the complete DES model for the process in Example 6.

Example 7 *The loan approval process in Example 6 contains five activities:* ⟨receive1⟩, ⟨invokeAssessor⟩, ⟨invokeApprover⟩, ⟨assign⟩, ⟨reply⟩. *The five activities are contained in a* ⟨flow⟩. *Six links,* ⟨receive_to_assess⟩, ⟨receive_to_approval⟩, ⟨assess_to_setMessage⟩, ⟨assess_to_approval⟩, ⟨approval_to_reply⟩, *and* ⟨setMessage_to_reply⟩, *connect the activities and change the concurrent orders to sequential orders between the activities. In this special case, there are actually no concurrent activities. Therefore, for clarity, the event caused by* ⟨flow⟩ *is not shown. Assume the approver may return an error message due to an unknown error. Below is the formal representation of the process (also reference to Figure 6).*

⟨receive1⟩ = $\langle\{x_0, x_1\}, \{received\}, \{t_1\}, \{x_0\}, \{x_1\}, \mathcal{C}\rangle$, with
$t_1 : (x_0 \wedge SoapMsg.type = MsgType) \xrightarrow{received} (x_1 \wedge request = SoapMsg)$, where $MsgType$ is a predefined message type. If the incoming message $SoapMsg$ has the predefined type, $request$ is initialized as $SoapMsg$.

⟨receive_to_assess⟩ = $\langle\{x_1, x_2\}, \{\epsilon\}, \{t_2\}, \{x_1\}, \{x_2\}, \mathcal{C}\rangle$, with
$t_2 : (x_1 \wedge request.amount < 1000) \xrightarrow{\epsilon} (x_2)$.

⟨receive_to_approval⟩ = $\langle\{x_1, x_3\}, \{\epsilon\}, \{t_3\}, \{x_1\}, \{x_3\}, \mathcal{C}\rangle$, with
$t_3 : (x_1 \wedge request.amount \geq 1000) \xrightarrow{\epsilon} (x_3)$.

⟨invokeAssessor⟩= $\langle\{x_2, x_4, x_5\}, \{invoked\_assessor, received\_risk\}, \{t_4, t_5\}, \{x_2\}, \{x_5\}, \mathcal{C}\rangle$ with
$t_4 : (x_2 \wedge InVar = request) \xrightarrow{invoked\_assessor} (x_4)$, and
$t_5 : (x_4) \xrightarrow{received\_risk} (x_5 \wedge OutVar = risk)$ where
$InVar$ and $OutVar$ are the input and output variables.

⟨assess_to_setMessage⟩ = $\langle\{x_5, x_6\}, \{\epsilon\}, \{t_6\}, \{x_5\}, \{x_6\}, \mathcal{C}\rangle$, with
$t_6 : (x_5 \wedge risk.level = low) \xrightarrow{\epsilon} (x_6)$.

⟨assess_to_approval⟩ = $\langle\{x_5, x_3\}, \{\epsilon\}, \{t_7\}, \{x_5\}, \{x_3\}, \mathcal{C}\rangle$, with
$t_7 : (x_5 \wedge risk.level = high) \xrightarrow{\epsilon} (x_3)$.

⟨invokeApprover⟩= $\langle\{x_3, x_7, x_8\}, \{invoked\_approver, received\_approval, received\_aplError\}, \{t_8, t_9, t_e\}, \{x_3\}, \{x_8\}, \mathcal{C}\rangle$ with
$t_8 : (x_3 \wedge InVar = request) \xrightarrow{invoked\_approver} (x_7)$, and

$t_9 : (x_7) \xrightarrow{received\_approval} (x_8 \wedge OutVar = approval)$, and
$t_e : (x_7) \xrightarrow{received\_aplError} (x_8 \wedge OutVar = errorMessage)$ where
$Invar$ and $OutVar$ are the input and output variables.

$\langle$assign$\rangle$: $\langle \{x_6, x_9\}, \{assigned\}, \{t_{10}\}, \{x_6\}, \{x_9\}, \mathcal{C} \rangle$ with
$t_{10} : (x_6) \xrightarrow{assigned} (x_9 \wedge approval.accept = yes)$

$\langle$setMessage_to_reply$\rangle$ = $\langle \{x_9, x_{10}\}, \{\epsilon\}, \{t_{11}\}, \{x_9\}, \{x_{10}\}, \mathcal{C} \rangle$, with
$t_{11} : (x_9) \xrightarrow{\epsilon} (x_{10})$.

$\langle$approval_to_reply$\rangle$ = $\langle \{x_8, x_{10}\}, \{\epsilon\}, \{t_{12}\}, \{x_8\}, \{x_{10}\}, \mathcal{C} \rangle$, with
$t_{12} : (x_8) \xrightarrow{\epsilon} (x_{10})$.

$\langle$reply$\rangle$: $\langle \{x_{10}, x_{11}\}, \{replied\}, \{t_{13}\}, \{x_{10}\}, \{x_{11}\}, \mathcal{C} \rangle$ with
$t_{13} : (x_{10} \wedge exists(approval)) \xrightarrow{replied} (x_{11} \wedge SoapMsg = approval)$, where
$SoapMsg$ is the message on the wire.

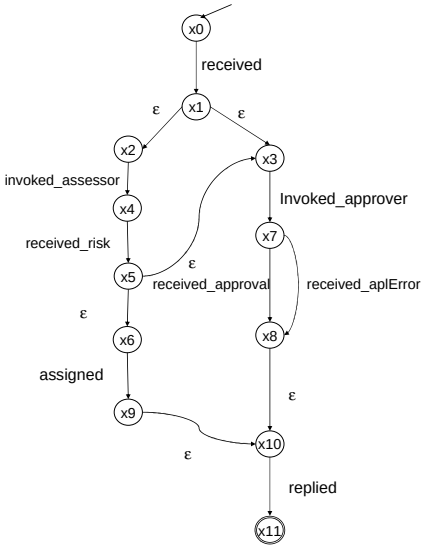


Figure 6: Automaton modeling loan approval process

5 Model-based Diagnosis for Web Service Processes

A Web service process can run down for many reasons. For example, a composed Web service may be faulty, an incoming message mismatches the interface, or the

Internet is down. The diagnosis task is to determine the Web services responsible for the exceptions. These Web services will be diagnosed faulty. In this paper, our major effort is on diagnosing business logic-related exceptions.

In our framework, $COMPS$ is made up of all the basic activities of the Web service process considered, and OBS is made up of the exceptions thrown and the events of the executed activities. These events can be obtained by the monitoring function of a BPEL engine. A typical correct model for an activity $\langle \mathsf{A} \rangle$ is thus:

$$\neg ab(A) \wedge \neg ab(A.input) \Longrightarrow \neg ab(A.output) \tag{4}$$

For facilitating diagnosis, the BPEL engine has to be extended for the following tasks: 1) record the events emitted by executed activities; 2) record the input and output SOAP messages; and 3) record the exceptions and trigger the diagnosis function when the first exception is received. ***Diagnosing*** is triggered on the first occurred exception [3]. The MBD approach developed relies on the following three steps with the techniques we introduced in the content above.

1) A prior process modeling and variable dependency analysis. All the variables in BPEL are global variables, i.e. they are accessible by all the activities. An activity can be regarded as a function that takes input variables and produces output variables. An activity has two kinds of relation to its input and output variables: defining and utilizing. We use $Def(A, V)$ and $Util(A, V)$ to present the relation that activity A defines variable V or utilizes V. An activity is normally a utilizer of its input variables, and is a definer of its output variables. This is similar to the view point of programming slicing, a technique in software engineering for software debugging (cf. Subsection 6.1). But BPEL can violate this relation by applying some business logic. For example, some variables, such as order ID and customer address, are not changeable after they are initialized in a business process. Therefore, a BPEL activity may be a utilizer of its output variables. In BPEL, it is defined in *correlation sets*. In this case, we use $Util(A, (V1, V2))$ to express that output $V2$ is correlated to input $V1$. In this case, Formula 4 can be simplified as:

$$\neg ab(A.input) \Longrightarrow \neg ab(A.output), \text{if } Util(A, (A.input, A.output)) \tag{5}$$

In Example 8, we give a table to summarize the variable dependency for the load approval process. This table can be obtained automatically from BPEL. The approach is not presented due to lack of space.

Example 8 *The variable dependency analysis for the loan approval process is in Table 1.*

2) Trajectories reconstruction from observations after exceptions are detected. As mentioned earlier, the observations are the events and exceptions when a BPEL process is executed. The events can be recovered from the log file in a BPEL engine. The observations are formed in an automaton. The possible trajectories of the process

[3] When a Web service engine supports multiple instances of a process, different instances are identified with a process ID. Therefore, diagnosis is based on the events for one instance of the process.

Variables	Parts	Definer	Utilizer
request	firstname	receive1	invokeAssessor, invokeApprover
	lastname	receive1	invokeAssessor, invokeApprover
	amount	receive1	invokeAssessor, invokeApprover
risk	level	invokeAssessor	
approval	accept	assign, invokeApprover	reply
error	errorCode	invokeApprover	

Table 1: The variable dependency analysis for the loan approval process.

are calculated by synchronizing the automaton of the observations with the automaton of the system description:

$$\mathsf{trajectories} = trajectories\ of\ SD||OBS \tag{6}$$

We do not require to record each event during the execution, but just enough to be able to identify the real trajectory of the process. This is very useful when some events are not observable and when there are too many events to record. Reference to Subsection 5.2 for more discussion.

Example 9 *In the loan approval example, assume that OBS=*{*received, invoked_assessor, received_risk, invoked_approver, received_aplErr*} *(as in Figure 7(a)).* $received\_aplErr$ *is an exception showing that there is a type mismatch in received parameters. We can build the trajectory of evolution as below, also shown in Figure 7(b).*

$$(x_0)\xrightarrow{received}(x_1)\xrightarrow{\epsilon}(x_2)\xrightarrow{invoked\_assessor}(x_4)\xrightarrow{received\_risk}(x_5)$$
$$\xrightarrow{\epsilon}(x_3)\xrightarrow{invoked\_approver}(x_7)\xrightarrow{received\_aplErr}(x_8)$$

3) Accountability analysis for mode assignment

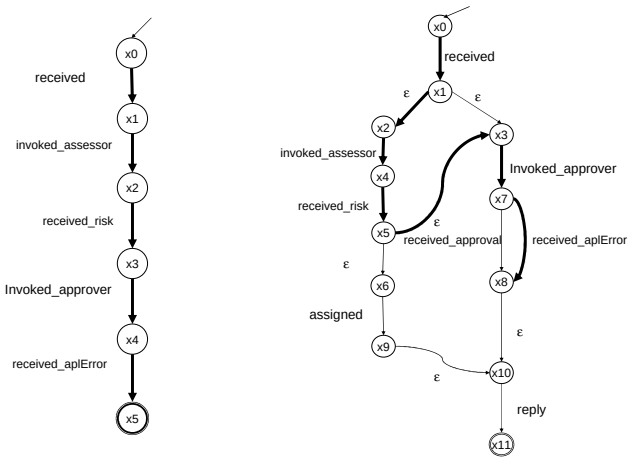
Not all the activities in a trajectory are responsible for the exception. As a software system, the activities connect to each other by exchanging variables. Only the activities which change the attributes within a variable can be responsible for the exception.

Assume that activity A generates exception e_f, and t is a trajectory ending at A. The responsibility propagation rules are (direct consequences of the contraposition of Formula 4 and 5):

$$e_f \in \Sigma_A \vdash ab(A) \vee \bigvee\{ab(A.InVar.part)|A.InVar.part \in A.InVar\} \tag{7}$$

$\forall A_i, A_j \in t, A_j \neq A_i$, A_j is the only activity between A_j and A_i such that $Def(A_j, A_i.InVar.part)$,

$$ab(A_i.InVar.part) \vdash ab(A_j) \vee \bigvee\{ab(A_j.InVar.part)|A_j.InVar.part \in A_j.InVar\} \tag{8}$$



(a) (b)

Figure 7: (a) the observations; (b) the loan approval process evolution trajectory up to the exception.

The first rule in (7) states that if an activity A generates an exception e_f, it is possible that activity A itself is faulty, or any part in its $A.InVar$ is abnormal. Notice a variable is a SOAP message which has several parts. $A.InVar.part$ is a part in $A.InVar$[4]. The second rule in (8) propagates the responsibility backwards in the trajectory. It states that an activity $A_j \in t$ that defines a part of $A_i.InVar$ which is known as faulty could be faulty; and its inputs could also be faulty. If there are several activities that define a part of $A_i.InVar$, only the last one counts, because it overrides the changes made by the other activities, i.e. A_j is the last activity "between" A_j and A_i that defines $A_i.InVar$, as stated in (8). After responsibility propagation, we obtain a *responsible set* of activities $RS = \{A_i\} \subseteq t$.

The set $CO = \{A\} \cup \{A_i | A_i \in RS\}$ is a conflict set, because if all the components in CO are correct, there should be no exceptions. Then a diagnosis is any of A or A_i in the responsible set is faulty:

$$\{D_f\} = \{\{A\}\} \cup \{\{A_i\} | A_i \in RS\} \tag{9}$$

Each D_f is a single fault diagnosis and the result is the disjunct of the D_f. The algorithm is as following. Lines 1-2 apply rule (7). Lines 3-8 apply rule (8). This algorithm checks each activity in t. Therefore the complexity of this algorithm is $O(|t|)$.

Example 10 *For the loan approval example, we have the trajectory as in Example 9. We do the responsibility propagation. As* $invokeApprover$ *generates the exception,*

[4]Sometimes, the exception returns the information about the part $A.InVar.part$ is faulty. Then this rule is simplified.

Algorithm 1 Calculate Diagnosis for a Faulty Web Service Process

INPUT: A_0 - the activity generating the exception.
t - a list of activities in a reserved trajectory ending at A_0, taken in reverse order with A_0 excluded.
Variables: V - a list of faulty variable parts, initialized as $\{\}$.
OUTPUT: D - the list of all possible faulty activities, initialized as $\{A_0\}$.
Notes about the algorithm: 1) list.next() returns the first element of a list; list.add(element) adds an element at the end of the list; list.remove(element) removes an element from the list. 2) Activity A has a list of input variables $A.InVars$ and output variables $A.OutVars$. 3) a variable var has a list of parts $var.Parts$.

```
1: for each variable var in A_0.InVars do
2:    V.add(var.Parts)
3: while A = t.next()! = null do
4:    if ∃p ∈ V, Def(A, p) then
5:       D.add(A)
6:       V.remove(v)
7:       for each variable var in A.InVars do
8:          V.add(var.Parts)
9: return D
```

according to Formula (7), $invokeApprover$ is possibly faulty. Then its input $request$ is possibly faulty. Among all the activities $\{receive1, invokeAssessor, invokeApprover\}$ in the trajectory, $receive1$ defines $request$, $invokeAssessor$ and $invokeApprover$ utilize $request$. Therefore, $receive1$ is possibly faulty, according to Formula (8). $receive1$ is the first activity in the trajectory. The propagation stops. The diagnosis is:

$$\{D_f\} = \{\{receive1\}, \{invokeApprover\}\}$$

Example 10 has two single faults $\{receive1\}$ and $\{invokeApprover\}$ for the exception $received\_aplErr$, which means either the activity ⟨receive1⟩ or ⟨invokeApprover⟩ is faulty. In an empirical way, an engineer may associate only one fault for an exception. But our approach can find all possibilities. Second, if we want to further identify which activity is indeed responsible for the exception, we can do a further test on the data. For example, if the problem is wrong data format, we can verify the data format against some specification, and then identify which activity is faulty.

5.1 Multiple Exceptions

There are two scenarios where multiple exceptions can happen. The first scenario is the chained exceptions when one exception causes the others to happen. Normally the software reports this chained relation. We need to diagnose only the first occurred exception, because the causal relations for other exceptions are obvious from the chain.

The second scenario is the case when exceptions occur independently, e.g. two paralleled branches report exceptions. As the exceptions are independent, we diagnose

each exception independently, the synthesis diagnoses are the union of all the diagnoses. Assume the minimal diagnoses for exception 1 are $\{D_i^1\}$, where $i \in [1, \ldots, n]$, and the minimal diagnoses for exception 2 are $\{D_j^2\}$, where $j \in [1, \ldots, m]$, the synthesis diagnoses are any combinations of D_i^1 and D_j^2: $\{D_i^1 \cup D_j^2 | i \in [1, \ldots, n], j \in [1, \ldots, m]\}$.

What interests us most is the synthesis of the minimal diagnoses. So, we remove the $D_i^1 \cup D_j^2$ that are supersets of other ones. This happens only if at least one activity is common to $\{D_i^1\}$ and $\{D_j^2\}$, giving rise to a single fault that can be responsible for both exceptions. Such activities are thus most likely to be faulty (single faults being preferred to double faults).

5.2 Without Full Observability

Equation 6 can recover trajectories from OBS. Actually if we can record all the events in a model, trajectories are equal to OBS. It is a trivial case. The problem occurs when we do not have full observability. For example, a third party BEPL engine does not allow us to record all the events crucial for diagnosis, or the process is too large to record every event. Equation 6 gets all the possible trajectories satisfying OBS. Therefore, this method can deal with missing events. At the meantime, if there are multiple trajectories satisfying OBS, the diagnoses are the union of the diagnoses obtained from all the trajectories. This can result in a larger number of diagnoses, i.e. diagnosis is not precise.

It is a trade off between observability and diagnosability. Increasing observability, i.e. observing more events, can result in more precise diagnosis, while increasing the observing cost. It is our future work to study the minimal observables for diagnosing a fault.

5.3 Offline Diagnosability Analysis

Diagnosability analysis is normally conducted offline without executing the processes. We do not touch diagnosability analysis problems in this paper. But diagnosability is related to the method of diagnosis. Assuming an exception at a place in a BPEL process, diagnosability analysis of this exception involves to calculate all the trajectories from the process entry point to the assumed exception and find diagnoses on each trajectory. The method is similar as the three steps in Section 5, just the second step is replaced by a graph traverse algorithm to compute all the trajectories between two nodes on the graph formed by the automaton model.

5.4 Multiple Trajectories

Lack of full observability and offline diagnosability analysis can cause multiple trajectories. Assume trajectories $\{t_1, ..., t_n\}$. Using our diagnosis algorithm, each trajectory t_i has conflict set CO_i. But as the trajectories are the possible execution paths, they do not occur at the same time, the conflict sets are not all contradictory at the same time. Indeed only one of these trajectories, even if which one is unknown, really happened. In this case, we do not have to use hitting set algorithm to compute diagnoses. We

define simply the synthesis diagnoses as the disjunction of all the diagnoses, $\vee\{D_i\}$, which means diagnoses are in any of $\{D_i\}$.

5.5 Obtaining the Dependency Table

The variable dependency table can be automatically constructed from BPEL. Regard a BPEL activity $\langle \mathsf{A} \rangle$ as a function $OutVar = f_A(InVar)$. Then $\langle \mathsf{A} \rangle$ is the utilizer of $InVar$ and definer of $OutVar$. Before, we have defined $\langle \mathsf{A} \rangle$ as an automaton. Then $InVar$ is the variables used in s_o and $Outvar$ is the variables used in s_f.

Due to some business logic, some variables, such as order ID and customer address, are not changeable after they are initialized in a business process. BPEL uses *correlation set* to define that two variables are identical in values. The correlation set is referenced by an activity. When an activity has a correlation set within its scope, the correlation indicates if this activity initiates the variables by setting the attribute `initiate`. If `initiate` is "yes", this activity is the definer for both of the variables, otherwise, this activity is the utilizer for both of the variables.

5.6 Implementation

There are many BPEL engines in the market. We extended ActiveBPEL (Active Endpoint, 2007), an open source from Active Endpoints, to implement our diagnosis mechanism. ActiveBPEL allows us to record every executed activity and messages in the execution. These activities and messages are the observations during execution and they correspond to a subset of the events and states in our formal model. Therefore, from the synchronization of the observations and the formal model result the execution trajectories. The diagnosis function is a java package that is invoked when an exception is caught. It takes the observations and the dependency table as inputs, calculates the trajectories and uses Algorithm 1 to calculate diagnoses.

6 Related Work and Discussion

6.1 A Brief Comparison to Program Slicing

Program slicing is a well known technique in software engineering for software debugging (Weise, 1984). If we have a specific program Π, a location within this program $\#n$ (n is a number given to a line), and a variable x, then a slice is itself a program that is obtained from the original program by removing all statements that have no influence on the given variable at the specified position. Since slices are usually smaller than the original program they focus the user's attention on relevant parts of the program during debugging. Slices can be computed from Program Dependence Graph (PDG) (Ottenstein & Ottenstein, 1984) as a graph reachability problem. A PDG G_Π for a program Π is a direct graph. The vertices of G_Π represent assignment statements and control predicates that occur in program Π. In addition G_Π includes the distinguished *entry* vertex. The edges of the graph represent either control or data dependencies. Given a criterion $\langle n, x \rangle$, the slice is computed in two steps. First, the vertex v representing the

last program position before n where variable x is defined must be localized. Second, the algorithm collects all vertices that can reach v via a control or flow dependency edge. The statements represented by the collected vertices (including vertex v) are equal to the program slice for Π.

Wotawa has discussed the relationship between MBD based debugging and program slicing (Wotawa, 2002). In his work, each statement in a program is viewed as a component with a mode, inputs and outputs. The logic representation of a statement $\#n$ is $\neg ab(n) \rightarrow out(n) = f(in(n))$, i.e. if $\#n$ is not faulty, the output $out(n)$ is a function of the input $in(n)$ according to the syntax of the program. He observed that the strongly connected components in the PDG have an influence one each other. Only if all the components are not faulty, the super component composed by these components is not faulty. He defined a dependency model whose nodes are the strongly connected components in the PDG and added a logic rule to describe the relation between the super component and the components within it. Assume $\{s_1, s_2, ..., s_n\}$ are strongly connected and the name of the super component is SC, then the rule is $\neg ab(s_1) \wedge ... \wedge \neg ab(s_n) \rightarrow \neg ab(SC)$. With the additional rule, logic deduction can more precisely identify the faulty components. Under this kind of modeling, slices of a single variable are equivalent to conflicts in MBD. And MBD and program slicing should draw equivalent conclusions on which statements are faulty.

We consider that diagnosing Web service processes is not equivalent to program debugging. First, we are interested in the faults due to the unknown behavior of the external Web services and due to the interaction between Web services. We assume that the Web service processes are described correctly in BPEL or a Web service process description language. This implicitly excludes the structured activities to be faulty. This is equivalent to consider only data dependency in program slice. Second, though Web service process description languages are like programs, they are simpler than programs. For example, they do not use pointers or other complicated data structures as in programs, and they do not use `Goto` and its other restricted forms as in unstructured program. This makes it possible that diagnosing Web service processes can be simpler than diagnosing programs.

The diagnosis method developed in this paper can be compared to dynamic slicing introduced in (Korel & Laski, 1988). Similar to our method, dynamic slicing considers the bugs should be within the statements that *actually* affect the value of a variable at a program point for a *particular* execution of the program. Their solution, following after Weiser's static slicing algorithm, solves the problem using data-flow equations, which is also similar to the variable dependency analysis presented in this paper, but not the same. An external Web service can be seen as a procedure in a program, with unknown behavior. For a procedure, we normally consider the outputs brought back by a procedure are generated according to the inputs. Therefore, in slicing, the outputs are considered in the definition set (the set of the variables modified by the statement). For Web services, we can know some parts in SOAP response back from a Web service should be unchanged, e.g. the name and the address of a client. This relation is defined as correlation set. We should point out that the variable dependency analysis in this paper is different from slicing. As a consequence, the diagnosis obtained from MBD approach in this paper can be different from slicing, and actually more precise.

As MBD approach can integrate more business logic into its model, it is less rigid

than slicing. In this sense, MBD is more business oriented, not program oriented, which makes it more suitable for diagnosing Web service processes than slicing.

6.2 MBD in Diagnosing Component-based Software

Besides Wotawa's work mentioned above, some other people have applied MBD on diagnosing component-based software systems. We found that when diagnosing such systems, the modeling is rather at the component level than translating lines of statements into logic representations. Grosclaude in (Grosclaude, 2004) used a formalism based on Petri nets to model the behaviors of component-based systems. It is assumed that only some of the events are monitored. The history of execution is reconstructed from the monitored events by connecting pieces of activities into possible trajectories. Console's group is working towards the same goal of monitoring and diagnosing Web services like us. In their paper (Ardissono et al., 2005), a monitoring and diagnosing method for choreographed Web service processes is developed. Unlike BPEL in our paper, choreographed Web service processes have no central model and central monitoring mechanism. (Ardissono et al., 2005) adopted grey-box models for individual Web services, in which individual Web services expose the dependency relationships between their input and output parameters to public. The dependency relationships are used by the diagnosers to determine the responsibility for exceptions. This abstract view could be not sufficient when dealing with highly interacting components. More specifically, if most of the Web services claim too coarsely that their outputs are dependent on their inputs, which is correct, the method in (Ardissono et al., 2005) could diagnose almost all the Web services as faulty. Yan *et al.* (Yan, Pencolé, Cordier, & Grastien, 2005) is our preliminary work to the present one, focusing on Web service modeling using transition systems. The major work in this paper is to complete the monitoring and diagnosis methods and present the diagnosis algorithm. The syntax of modeling in this paper is improved from (Yan et al., 2005) with simplified representation of states and explicit definition of constraints. As a result, the model for a process can be more readable and a slightly fewer states. This paper is also self-contained with MBD background and discussions on fault management tasks for Web service processes.

6.3 Related Work in Web Service Monitoring, Modeling and Composition

Several groups of researchers work on Web service monitoring frameworks. (Baresi, Ghezzi, & Guinea, 2006) proposes BPEL^2 which is the original BPEL with monitoring rules. Monitoring rules define how the user wants to oversee the execution of BPEL. But (Baresi et al., 2006) does not specify the monitoring tasks. (Mahbub & Spanoudakis, 2004) proposes a framework for monitoring requirements of BPEL-based service compositions. Their approach uses event calculus for specifying the requirements that must be monitored. Requirements can be behavioral properties of the coordination process or assumptions about the atomic or joint behavior of the deployed services. Events, produced by the normal execution of the process, are stored

in a database and the runtime checking is done by an algorithm based on integrity constraint checking in temporal deductive databases. These frameworks can be used for recording the events and messages used for diagnosis.

In addition to automata used in this paper, Petri nets and process algebra are also used as formal models for Web service processes. For example, (Salaün, Bordeaux, & Schaerf, 2004; Ferrara, 2004; Viroli, 2004) map BPEL into different Process Algebra; (Ouyang et al., 2005; Schmidt & Stahl, 2004) present different semantics of BEPL in Petri nets; (Fisteus, Fernández, & Kloos, 2004; Foster, Uchitel, Magee, & Kramer, 2003; Fu, Bultan, & Su, 2004) use automata to model BPEL for verification. These models have similar expression power and similar reasoning or computing techniques.

Web service composition techniques are relevant to this paper because they generate new Web service processes. AI planning methods are the most commonly used techniques for Web service composition. (Narayanan & McIlraith, 2002) starts from DAML-S descriptions and automatically transforms them into Petri nets. Other works, as (Berardi, Calvanese, De Giacomo, Lenzerini, & Mecella, 2003; Lazovik, Aiello, & Papazoglou, 2003; Pistore, Traverso, Bertoli, & Marconi, 2005), rely on transition rules systems. (Rao & Su, 2004) is a survey paper on automated Web service composition methods. Re-planning is relevant to this paper because it can be used to modify the Web service processes for fault recovery. (Canfora, Penta, Esposito, & Willani, 2005) presents a re-planning technique based on slicing techniques. When the estimated QoS metrics are not satisfied, the re-planning selects other Web services to replace the ones in the process.

7 Conclusion

Web services are the emergent technology for business process integration. A business process can be composed of distributed Web services. The interactions among the Web services are based on message passing. To identify the Web services that are responsible for a failed business process is important for e-business applications. Existing throw-and-catch fault handling mechanism is an empirical mechanism that does not provide sound and complete diagnosis. In this paper, we developed a monitoring and diagnosis mechanism based on solid theories in MBD. Automata are used to give a formal modeling of Web service processes described in BPEL. We adapted the existing MBD techniques for DES to diagnose Web service processes. Web service processes have all the features of software systems and do not appear to function abnormally until an exception is thrown and they are stopped, which makes the diagnosis principle different from diagnosing physical systems. The approach developed here reconstructs execution trajectories based on the model of the process and the observations from the execution. The variable dependency relations are utilized to deduce the actual Web services within a trajectory responsible for the thrown exceptions. The approach is sound and complete in the context of modeled behavior. A BPEL engine can be extended for the monitoring and diagnosis approach developed in this paper.

References

Active Endpoint. (2007). *www.active-endpoints.com/active-bpel-engine-overview.htm.*

Andrews, T., Curbera, F., Dholakia, H., Goland, Y., & et.al. (2003). *Business process execution language for web services (bpel4ws) 1.1.* (*ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf*, retrieved April 10, 2005)

Ardissono, L., Console, L., Goy, A., Petrone, G., Picardi, C., & Segnan, M. (2005). Cooperative model-based diagnosis of web services. In *Proceedings of the 16th international workshop on principles of diagnosis (DX-2005)* (pp. 125–132).

Baresi, L., Ghezzi, C., & Guinea, S. (2006). Towards self-healing compositions of services. In B. J. Krämer & W. A. Halang (Eds.), *Contributions to ubiquitous computing* (Vol. 42). Springer.

Baroni, P., Lamperti, G., Pogliano, P., & Zanella, M. (1999). Diagnosis of large active systems. *Artificial Intelligence*, *110*(1), 135–183.

Berardi, D., Calvanese, D., De Giacomo, G., Lenzerini, M., & Mecella, M. (2003). Automated composition of e-services that export their behavior. In *Proceedings of the 1st int. conf. on service-oriented computing (icsoc'03)* (p. 43-58).

Canfora, G., Penta, M. D., Esposito, R., & Willani, M. L. (2005). Qos-aware replanning of composite web services. In *Proceedings of ieee international conference on web serices.*

Cordier, M.-O., & Thiébaux, S. (1994). Event-based diagnosis for evolutive systems. In *Proceedings of the fifth international workshop on principles of diagnosis(DX'94)* (pp. 64–69).

Ferrara, A. (2004). Web services: a process algebra approach. In *Proceedings of the 2nd international conference on service oriented computing (icsoc)* (p. 242-251). New York, NY, USA: ACM Press.

Fisteus, J., Fernández, L., & Kloos, C. (2004). Formal verification of bpel4ws business collaborations. In K. Bauknecht, M. Bichler, & B. Prll (Eds.), *Proc. of 5th international conference e-commerce and web technologies (ec-web).*

Foster, H., Uchitel, S., Magee, J., & Kramer, J. (2003). Model-based verification of web service compositions. In *Proc. of eighteenth ieee international conference on automated software engineering (ase03).*

Fu, X., Bultan, T., & Su, J. (2004). Analysis of interacting bpel web services. In *Proc. of the 13th international world wide web conference (www'04).* ACM Press.

Grastien, A., Cordier, M.-O., & Largouët, C. (2004). Extending decentralized discrete-event modelling to diagnose reconfigurable systems. In *Proceedings of the fifteenth international workshop on principles of diagnosis (DX-04)* (pp. 75–80). Carcassonne, France.

Grastien, A., Cordier, M.-O., & Largouët, C. (2005). Incremental diagnosis of discrete-event systems. In *Proceedings of the sixteenth international workshop on principles of diagnosis (DX-05)* (pp. 119–124). Pacific Grove, California, USA.

Grosclaude, I. (2004). Model-based monitoring of software components. In *Proceedings of the 16th european conf. on artificial intelligence (ECAI'04)* (pp. 1025–1026).

Hamscher, W., Console, L., & de Kleer, J. (Eds.). (1992). *Readings in model-based diagnosis*. Morgan Kaufmann.

Korel, B., & Laski, J. (1988). Dynamic program slicing. *Information Processing Letters*, *29(3)*, 155-163.

Lazovik, A., Aiello, M., & Papazoglou, M. (2003). Planning and monitoring the execution of web service requests. In *Proceedings of the 1st int. conf. on service-oriented computing (ICSOC'03)* (pp. 335–350).

Mahbub, K., & Spanoudakis, G. (2004). A framework for requirents monitoring of service based systems. In *Proceedings of the 2nd international conference on service oriented computing* (p. 84-93).

Narayanan, S., & McIlraith, S. (2002). Simulation, verification and automated compostion of web services. In *Proceedings of the eleventh international world wide web conference (www-11)* (p. 77-88).

Ottenstein, K., & Ottenstein, L. (1984). The program dependence graph in software development environment. In *Acm sigsoft/sigplan software engineering symposium on practical software development environments* (p. 177-184).

Ouyang, C., Aalst, W. van der, Breutel, S., Dumas, M., Hofstede, A. ter, & Verbeek, H. (2005). *Formal semantics and analysis of control flow in ws-bpel* (Tech. Rep.). BPM Center Report BPM-05-13. (BPMcenter.org)

Pencolé, Y., & Cordier, M.-O. (2005). A formal framework for the decentralised diagnosis of large scale discrete event systems and its application to telecommunication networks. *Artificial Intelligence Journal*, *164*(1-2), 121–170.

Pencolé, Y., Cordier, M.-O., & Rozé, L. (2002). Incremental decentralized diagnosis approach for the supervision of a telecommunication network. In *Proceedings of 41th ieee conf. on decision and control (CDC'2002)* (pp. 435–440). Las Vegas, USA.

Pistore, M., Traverso, P., Bertoli, P., & Marconi, A. (2005). Automated composition of web services by planning at the knowledge level. In *Proceedings of the 19th international joint conference on artificial intelligence (ijcai-05)* (p. 1252-1260).

Rao, J., & Su, X. (2004). A survey of automated web service composition methods. In *Proceedings of the first international workshop on semantic web services and web process composition (swswpc)*.

Reiter, R. (1987). A theory of diagnosis from first principle. *Artificial Intelligence*, *32*(1), 57-95.

Salaün, G., Bordeaux, L., & Schaerf, M. (2004). Describing and reasoning on web services using process algebra. In *Proceedings of the second ieee int. conf. on web services (ICWS'04)* (pp. 43–51).

Sampath, M., Sengupta, R., Lafortune, S., Sinnamohideen, K., & Teneketzis, D. (1995). Diagnosability of discrete-event systems. *IEEE Transactions on Automatic Control*, *40*(9), 1555-1575.

Schmidt, K., & Stahl, C. (2004). A petri net semantic for bpel: validation and application. In *Proc. of 11th workshop on algorithms and tools for petri nets (awpn 04)* (p. 1-6).

Viroli, M. (2004). Towards a formal foundation to orchestration languages. In *Electronic notes in theoretical computer science 105* (p. 51-71). Elsevier.

Weise, M. (1984). Program slicing. *IEEE Transactions on Software Engineering*, *10(4)*, 352-357.

Wotawa, F. (2002). On the relationship between model-based debugging and program slicing. *Artificial Intelligence*, *135*, 125-143.

Yan, Y., Pencolé, Y., Cordier, M.-O., & Grastien, A. (2005). Monitoring web service networks in a model-based approach. In *3rd ieee european conference on web services (ecows05).* Växjö, Sweden: IEEE Computer Society.